# Intro to CRUD with PyMongo

## About Me

I’m a Branham Alum (Class of 2015) and went on to get a BS in Statistics from CalPoly SLO

My field of expertise is Statistics, Data Analytics, Data Science/ML

I’m also a coach here which is how Jin got me to sign up for this

## Prerequisites

- Python Basics
  - Basic data types (Lists, Dictionaries)
  - Loops and iteration
  - Importing modules
  - Some experience with what objects and functions are
    - Don’t need immense object-oriented experience

- Knowledge of JSON filetype, Jupyter, Google Colab, and PIP might be helpful but are not required

## Databases

- Databases are stored collections of data
  - Two main types of databases: structured and unstructured

- Data is king at the moment and no company does not have a database of some kind

- Databases are managed by Database Management Systems (DBMS)
  - The most common DBMS is Structured Query Language (SQL) or its variants
  - We will be working with MongoDB today

# What is CRUD?

CRUD is the acronym for the 4 database operations

- Create: Adding data to the database
- Read: Retrieving data from the database for use
- Update: Updating already present data in the database
- Delete: Removing data from the database

## MongoDB

- MongoDB is a NoSQL database
  - NoSQL means it isn't a variant of SQL and does not use the same style of syntax
    - This makes it way easier to learn!
    - Some people claim NoSQL is short for "not only structured query language"
- Incredibly popular DBMS at the moment and has tons of support and examples of how to use it in real applications
- We will be using PyMongo to access a MongoDB server through Python
- Uses a JSON structure so if you know that, you already get most of this
  - Technically it is BSON but that is not important

# MongoDB – Terms to know

**Client**: Basically, a server running MongoDB (in our case one I’m hosting on the cloud)

**Database**: Client’s have databases in them

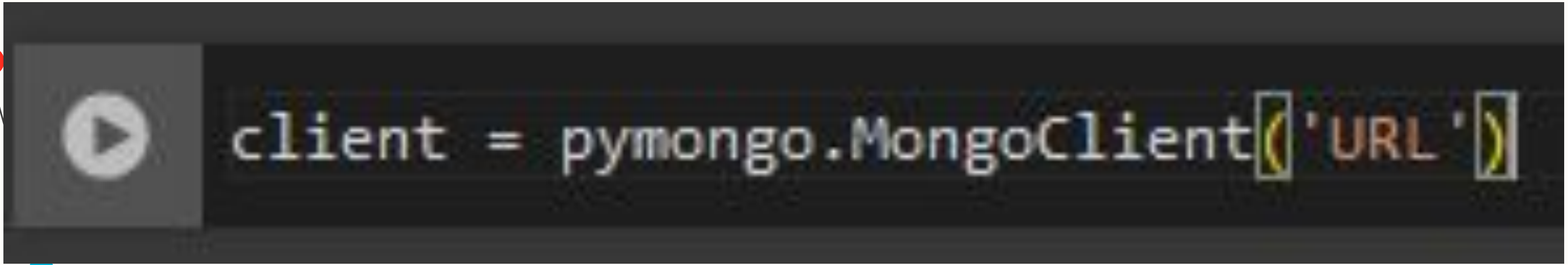
**Collection**: An individual set of data in a database, think an excel spreadsheet

**Document**: An individual entry in a collection, a single row in that spreadsheet

## Setup

1. Go to https://colab.research.google.com/
2. Upload > Upload the workshop.ipynb
3. Remove the ‘#’ from the first cell
4. Run the first cell ‘!pip install pymongo’
    a. Click the play button or hit Shift+Enter
5. Give it a moment to install the package

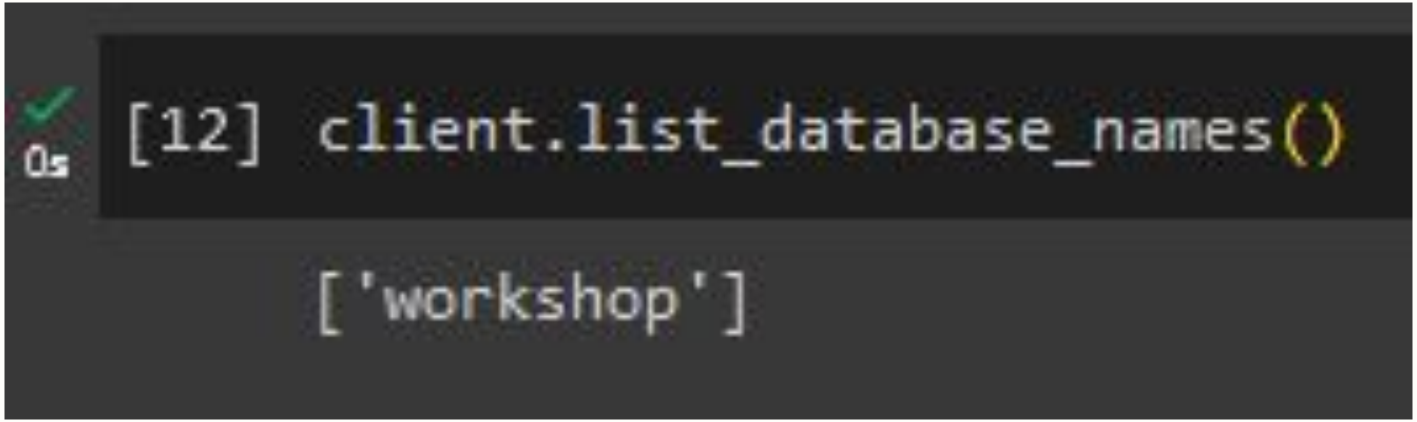
## Connecting to MongoDB



```
client = pymongo.MongoClient('URL')
```

# Connecting to MongoDB

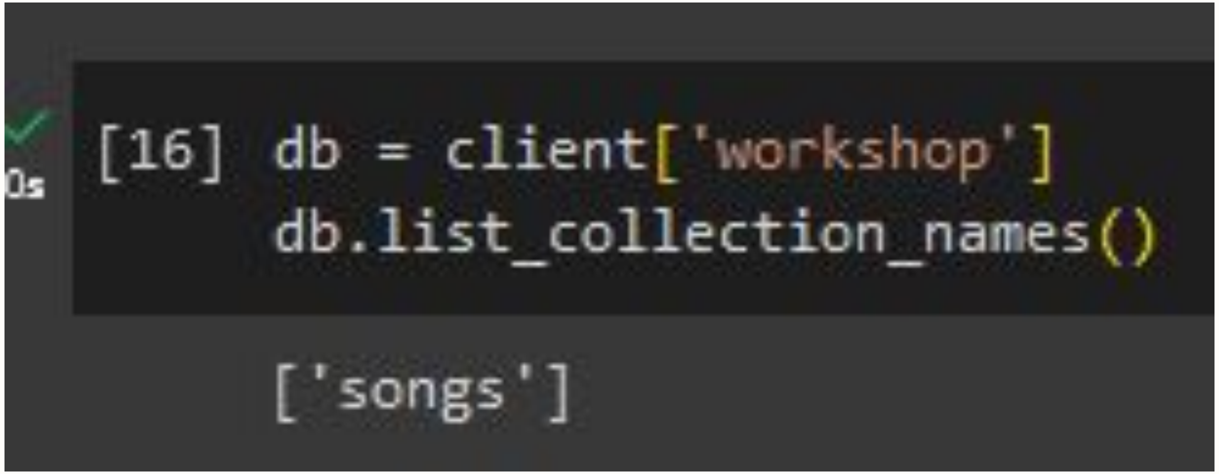
If your connection worked, you should be able to run the ‘**.list_database_names()**’ method and see an output similar to this:



```
0s  [12] client.list_database_names()

         ['workshop']
```

# Getting to our database

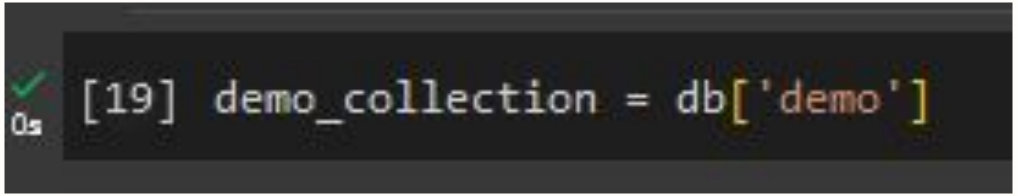
We can index to the database we will be using and list which collections are in that database by using the ‘**.list_collection_names()**’ method.



```
[16] db = client['workshop']
     db.list_collection_names()

     ['songs']
```

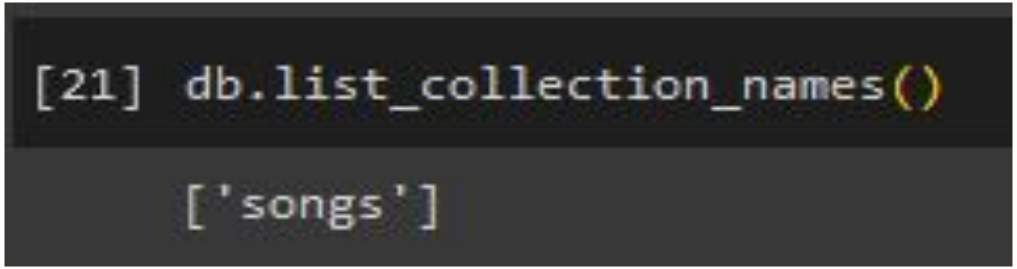
# Create – Making a collection

- Let's make a collection, MongoDB makes it super easy:



```
0s [19] demo_collection = db['demo']
```

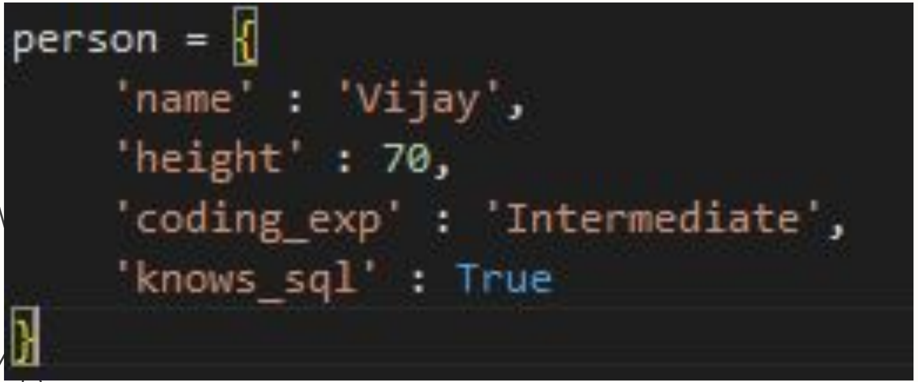
- The collection will not exist until you actually add something to it!



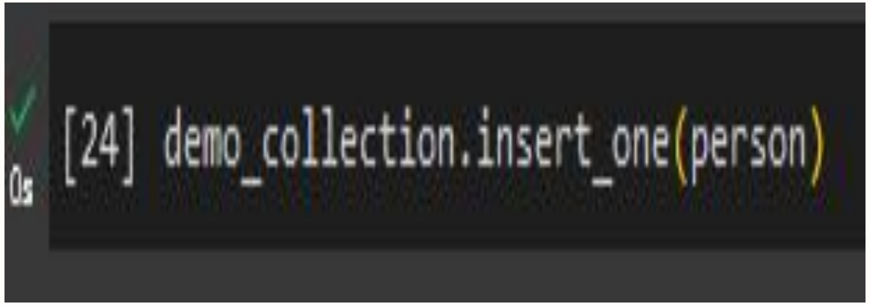
```
[21] db.list_collection_names()

     ['songs']
```

## Create – Making a document

1. First, let’s make a dictionary of our entry
2. Then, we can insert it using the ‘**.insert_one()**’ method



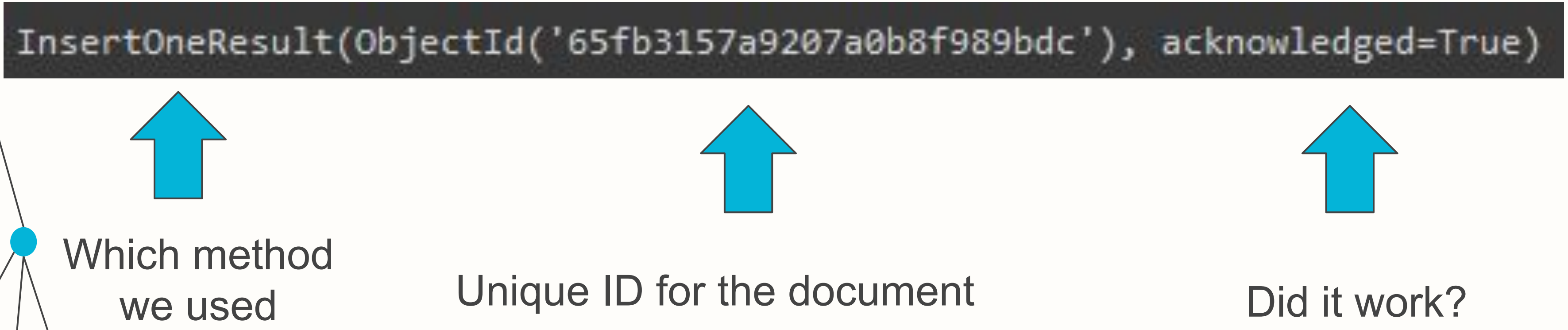
```
person = {
    'name' : 'Vijay',
    'height' : 70,
    'coding_exp' : 'Intermediate',
    'knows_sql' : True
}
```



```
[24] demo_collection.insert_one(person)
```

## Create – What just happened?

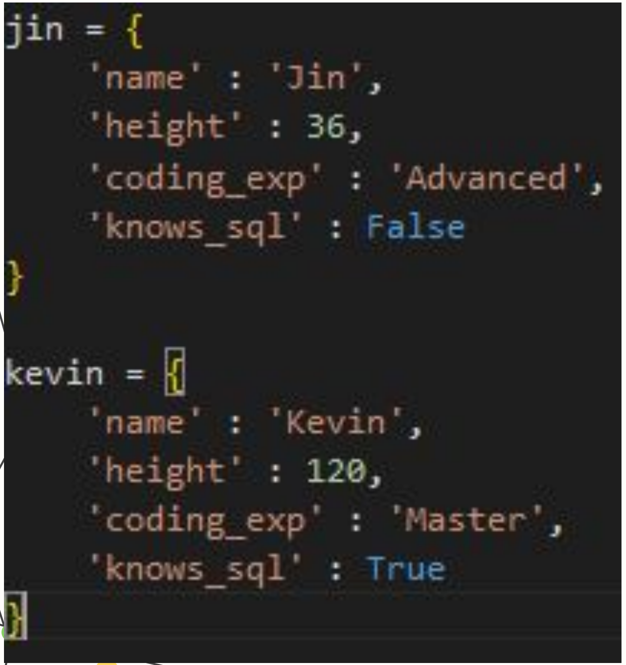
When we perform any action that changes the database (inserting, removing, modifying), we will get a response back from MongoDB telling us if it worked and sometimes other information.



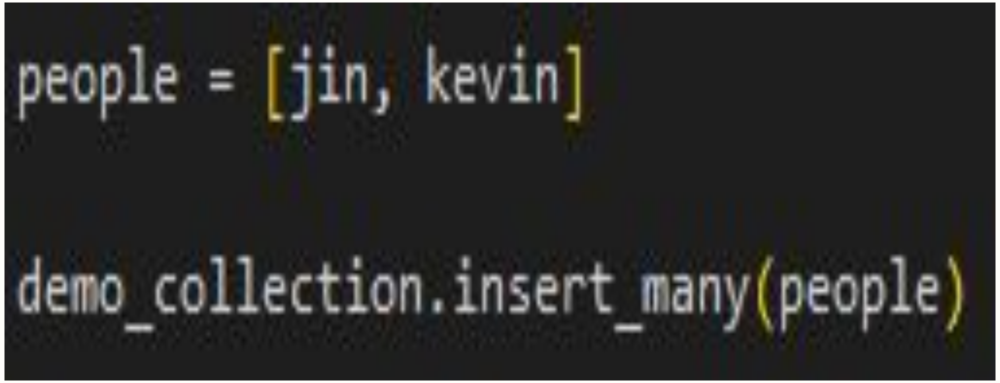
```
InsertOneResult(ObjectId('65fb3157a9207a0b8f989bdc'), acknowledged=True)
```

## Create – Adding more than one at a time

You can add multiple entries to the collection with the ‘**.insert_many()**’ function



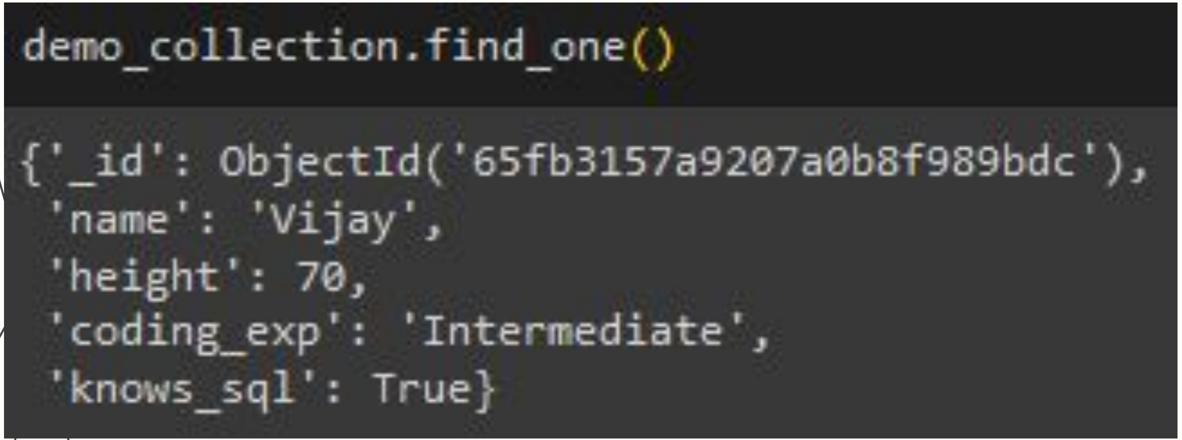
```
jin = {
    'name' : 'Jin',
    'height' : 36,
    'coding_exp' : 'Advanced',
    'knows_sql' : False
}

kevin = {
    'name' : 'Kevin',
    'height' : 120,
    'coding_exp' : 'Master',
    'knows_sql' : True
}
```



```
people = [jin, kevin]

demo_collection.insert_many(people)
```

## Read – Finding a single entry

Let’s check that our data is actually in the collection properly, first we’ll use the ‘**.find_one()**’ method.



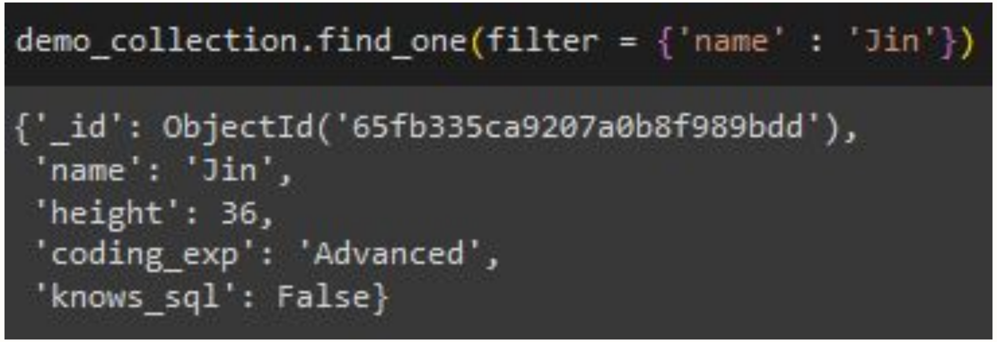
```
demo_collection.find_one()
```

```
{'_id': ObjectId('65fb3157a9207a0b8f989bdc'),
 'name': 'Vijay',
 'height': 70,
 'coding_exp': 'Intermediate',
 'knows_sql': True}
```

‘.find_one()’ only returns the FIRST entry that it finds!

## Read – Finding a specific entry

We can use the ‘**filter**’ parameter to find an entry that matches our criteria.



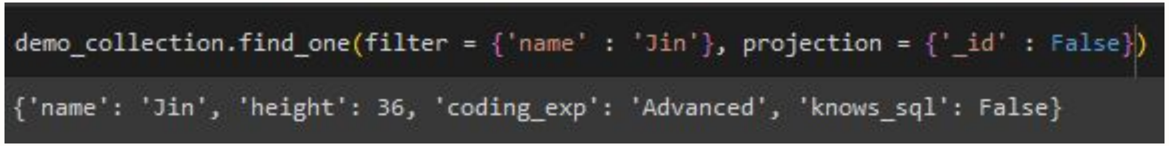
```
demo_collection.find_one(filter = {'name' : 'Jin'})
```

```
{'_id': ObjectId('65fb335ca9207a0b8f989bdd'),
 'name': 'Jin',
 'height': 36,
 'coding_exp': 'Advanced',
 'knows_sql': False}
```

# Read – Projections

This is nice but we have that gross looking '_id' that isn't really important to us.

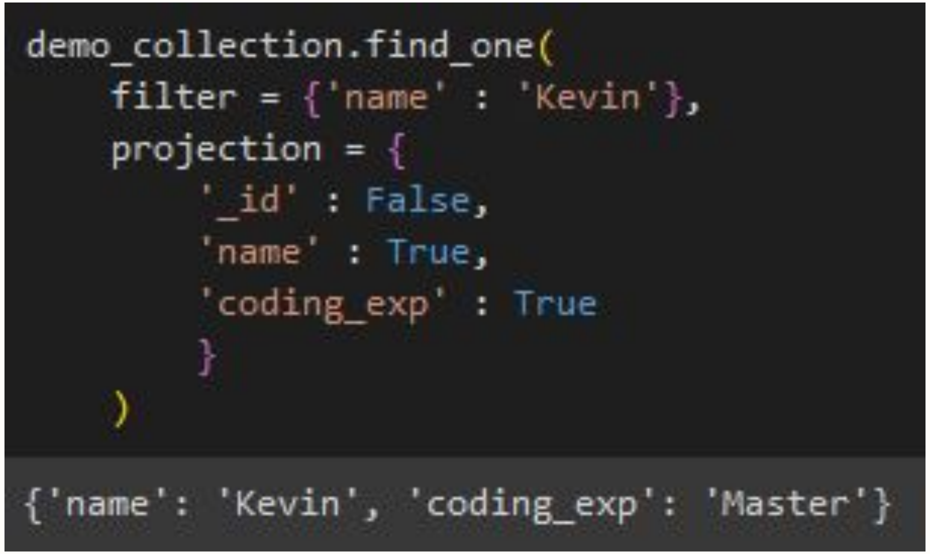
The '**projection**' parameter lets us limit which fields MongoDB returns!



```
demo_collection.find_one(filter = {'name' : 'Jin'}, projection = {'_id' : False})
{'name': 'Jin', 'height': 36, 'coding_exp': 'Advanced', 'knows_sql': False}
```

## Read – Projects II

We can also use projections to get only specific fields



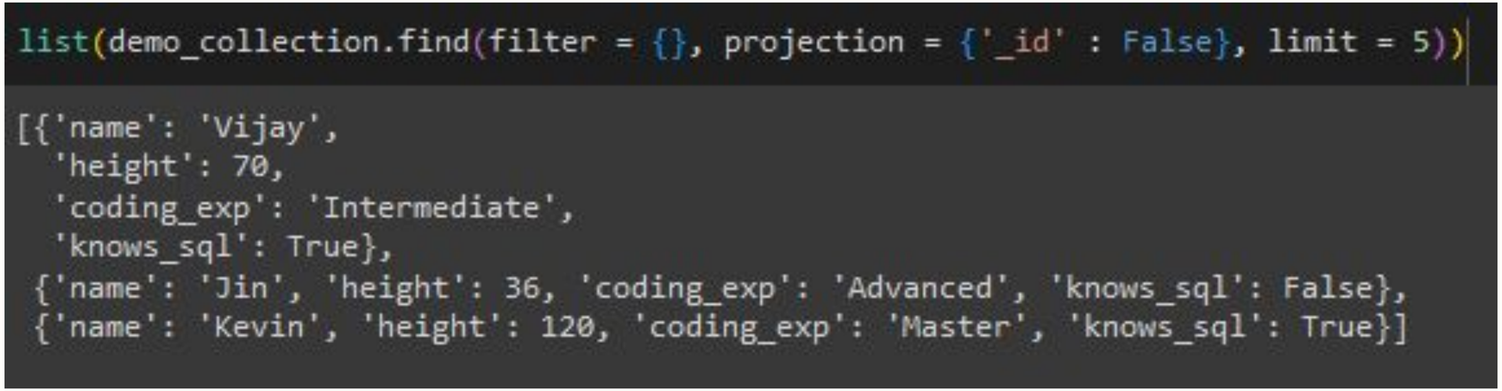
```
demo_collection.find_one(
    filter = {'name' : 'Kevin'},
    projection = {
        '_id' : False,
        'name' : True,
        'coding_exp' : True
        }
    )
```

```
{'name': 'Kevin', 'coding_exp': 'Master'}
```

# Read – Finding more than one entry

Just like with insert, we can use a separate method to find multiple entries

This method returns an object called a Cursor, to simplify things, we will convert that into a list directly.

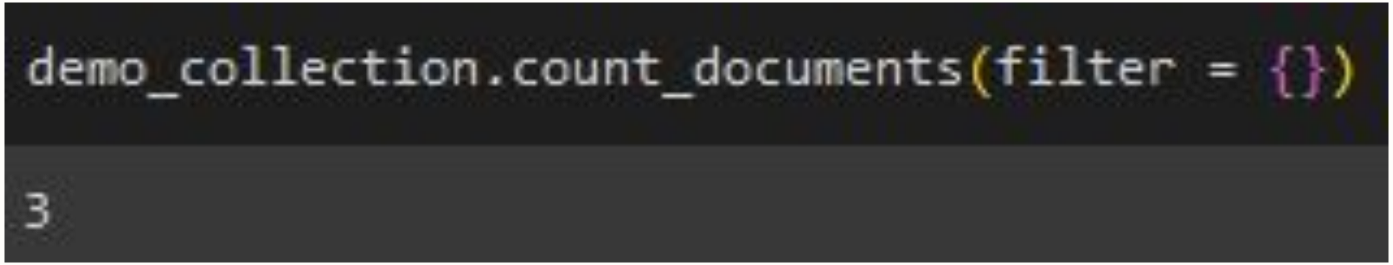


```
list(demo_collection.find(filter = {}, projection = {'_id' : False}, limit = 5))
```

```
[{'name': 'Vijay',
  'height': 70,
  'coding_exp': 'Intermediate',
  'knows_sql': True},
 {'name': 'Jin', 'height': 36, 'coding_exp': 'Advanced', 'knows_sql': False},
 {'name': 'Kevin', 'height': 120, 'coding_exp': 'Master', 'knows_sql': True}]
```

## Read – Checking the size of a collection

As a useful tip, the ‘**.count_documents()**’ method tells you how many documents are present.



```
demo_collection.count_documents(filter = {})
```

```
3
```

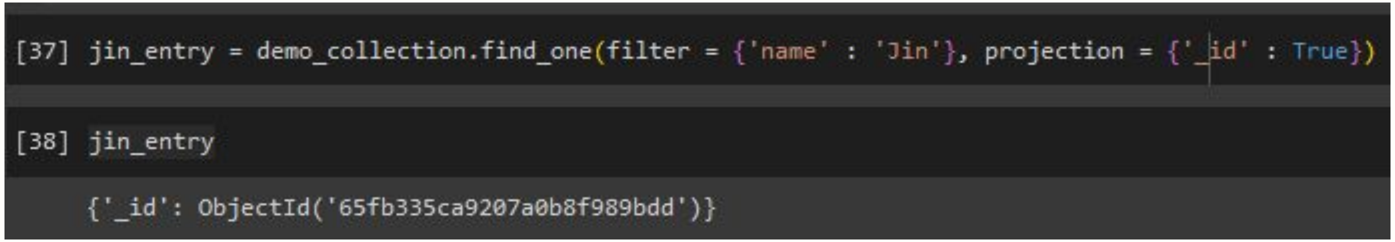
## Update – modifying existing fields

Like with insert and find, we can do this for just one entry or as many entries that fit a criteria.

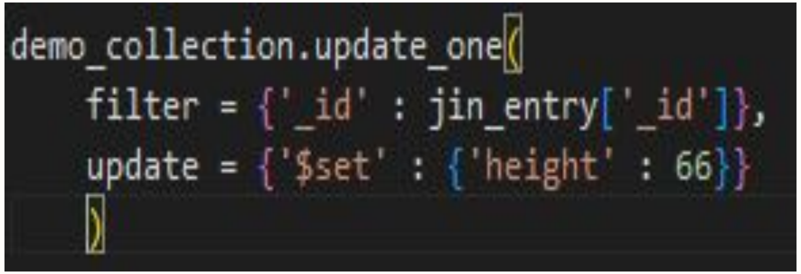
While updating (or removing) it is best to be as specific as possible! We will use the ObjectIDs to make sure we only change the exact entry we care about

To update we will use an **operator** to tell MongoDB what type of update we want to do, in our case '$set' to set a value of our choosing

## Updating



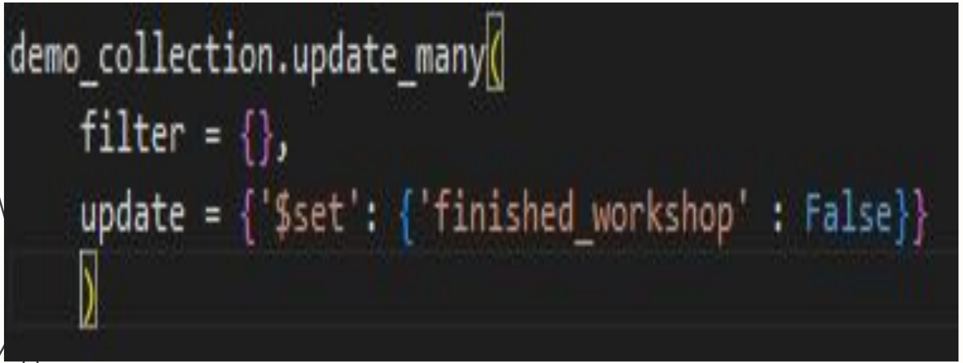
```
[37] jin_entry = demo_collection.find_one(filter = {'name' : 'Jin'}, projection = {'_id' : True})

[38] jin_entry

     {'_id': ObjectId('65fb335ca9207a0b8f989bdd')}
```



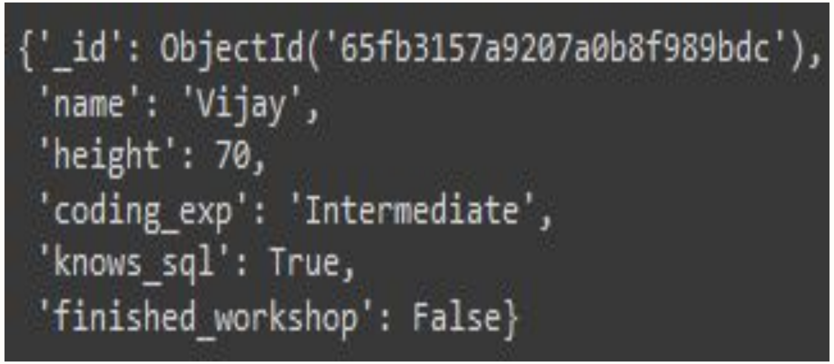
```
demo_collection.update_one(
    filter = {'_id' : jin_entry['_id']},
    update = {'$set' : {'height' : 66}}
    )
```

## Update – Adding fields

The ‘**$set**’ operator also works to add a new field entirely



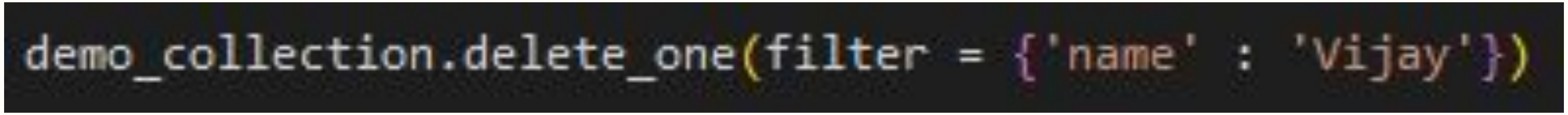
```
demo_collection.update_many(
    filter = {},
    update = {'$set': {'finished_workshop' : False}}
    )
```



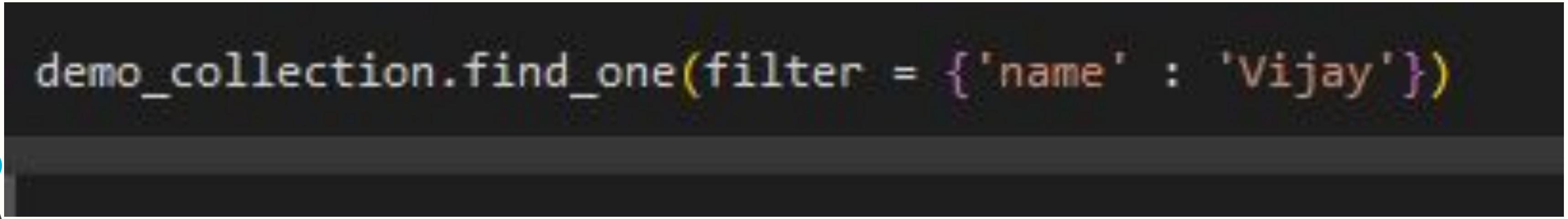
```
{'_id': ObjectId('65fb3157a9207a0b8f989bdc'),
 'name': 'Vijay',
 'height': 70,
 'coding_exp': 'Intermediate',
 'knows_sql': True,
 'finished_workshop': False}
```

# Delete – Removing entries

Just like before, we have methods to delete one or more entries at a time
For now, everyone can delete their own entry



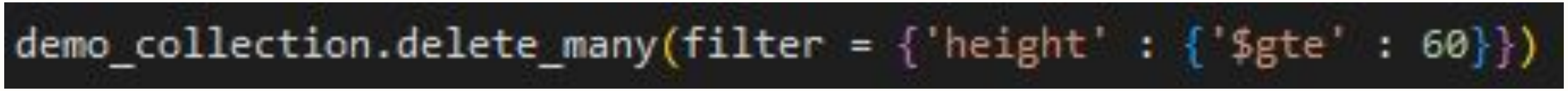
```
demo_collection.delete_one(filter = {'name' : 'Vijay'})
```



```
demo_collection.find_one(filter = {'name' : 'Vijay'})
```

## Delete – Removing multiple entries

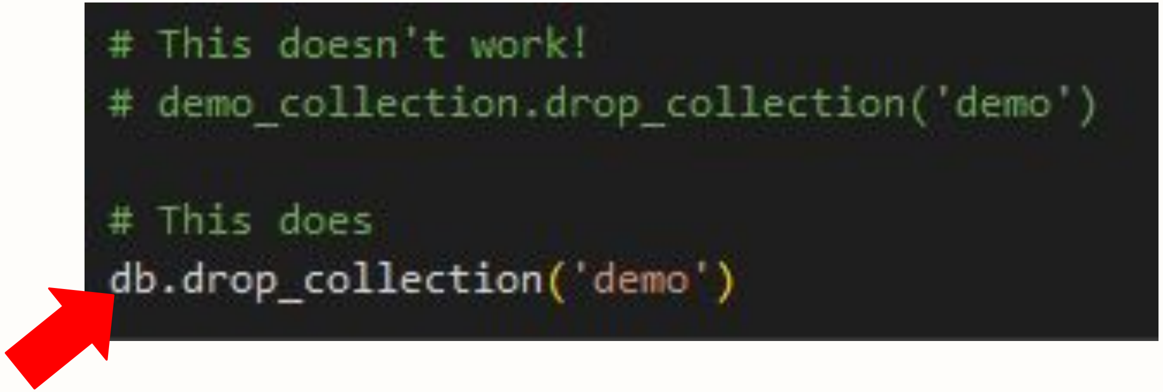
Deleting many is the same as before, just set a filter and it will delete everything that matches the filter



```
demo_collection.delete_many(filter = {'height' : {'$gte' : 60}})
```

## Delete – Deleting a collection

You can delete a collection by using the '**.drop_collection()**' method.
This method must be run on a Database **not** on a collection!



```
# This doesn't work!
# demo_collection.drop_collection('demo')

# This does
db.drop_collection('demo')
```

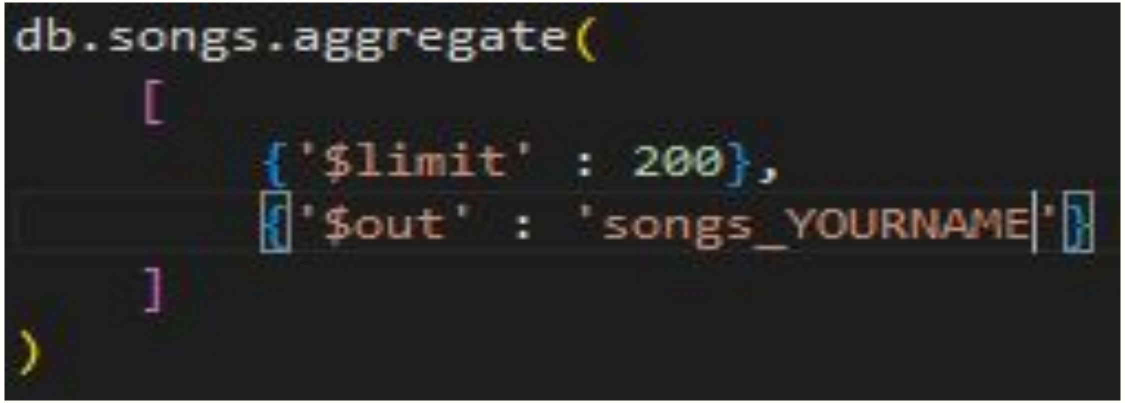
## CRUD

That’s all 4 of the CRUD operations at their most basic level.

In practice, these operations get a lot more complicated. We will go into a bit more depth, in particular with reads

## Prep

So we don't edit each other's collections, let's all make a copy of the '**songs**' collection that we can use on our own.



```
db.songs.aggregate(
        [
                {'$limit' : 200},
                {'$out' : 'songs_YOURNAME'}
        ]
)
```

## Update – Renaming fields

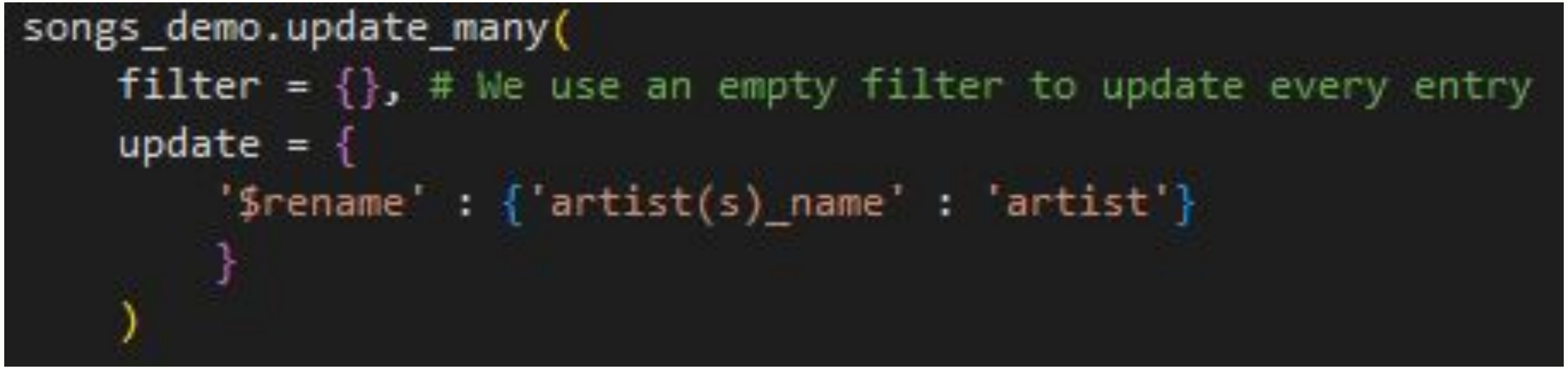
The field for artist’s name is super long and annoying to type out, so let’s fix it!

We can use the ‘**.update_many()**’ method and the ‘**$rename**’ operator to do this

The ‘**$rename’** operator works like this:

```
{‘$rename’ : {‘old_name’ : ‘new_name’}}
```

# Update – Example



```
songs_demo.update_many(
    filter = {}, # We use an empty filter to update every entry
    update = {
        '$rename' : {'artist(s)_name' : 'artist'}
        }
    )
```

## Remove

Unfortunately, our dataset includes Chris Brown so let’s get rid of him.

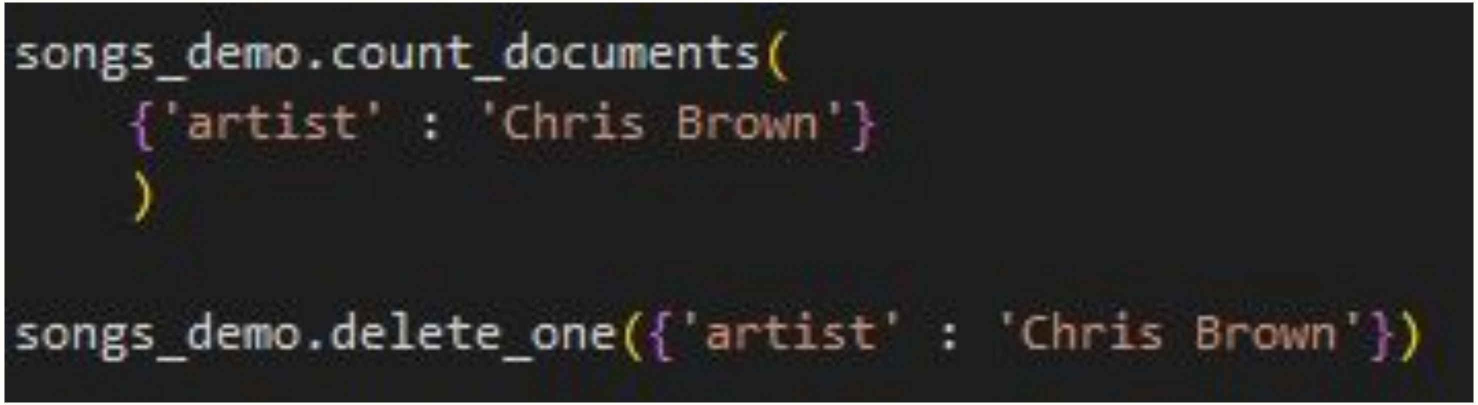
First, how many of his songs are in your collection?

Use the ‘**.count_documents()**’ method and a filter to find out

Once you’ve found the count, try removing him! (‘**.delete_many()**’/’**.delete_one()**’)

If you did it correctly, rerunning your count documents should return 0

## Removing – Example



```
songs_demo.count_documents(
    {'artist' : 'Chris Brown'}
    )

songs_demo.delete_one({'artist' : 'Chris Brown'})
```

## Read – With operators this time

As we saw with the update methods, operators use the '**$**' to tell MongoDB we are doing an operation.

The common operators are:

- **$gt, $lt, $gte, $lte**: Greater/less than (or equal to)
- **$eq, $neq**: Equal or Not Equal
- **$in, $nin**: in or not in a list of values

## Read – Compound Queries

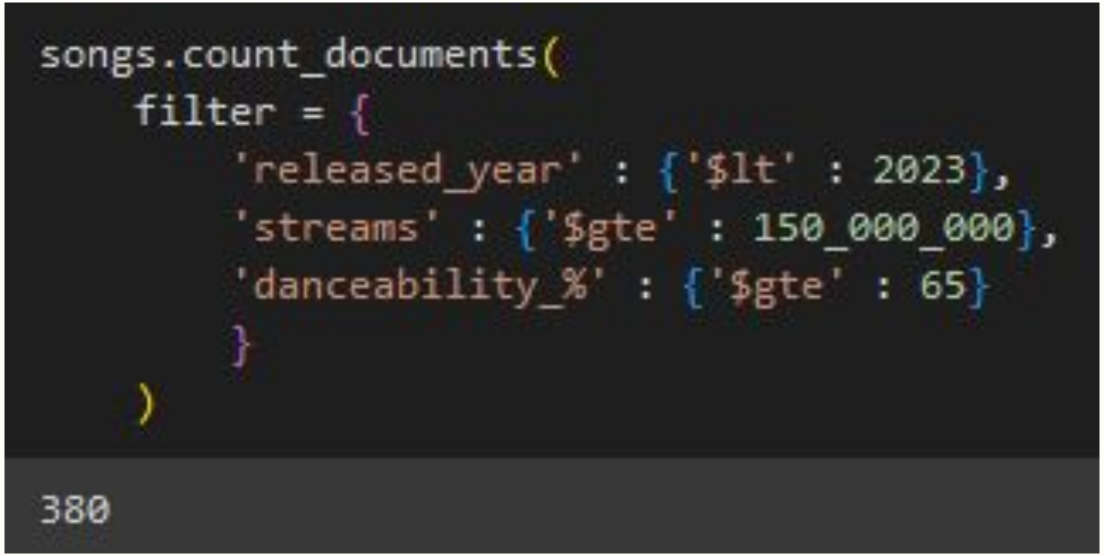
Now that we have our collection, let’s try using a few filters and methods together

We’ll count how many songs were released before 2023, streamed at least 150,000,000 times, and are at least 65% danceable.

We will need to use two operators for this:

- **$lt** for less than 2023
- **$gte** for greater than or equal to 150,000,000 and 65

## Read – Example Query



```
songs.count_documents(
    filter = {
        'released_year' : {'$lt' : 2023},
        'streams' : {'$gte' : 150_000_000},
        'danceability_%' : {'$gte' : 65}
        }
    )
```

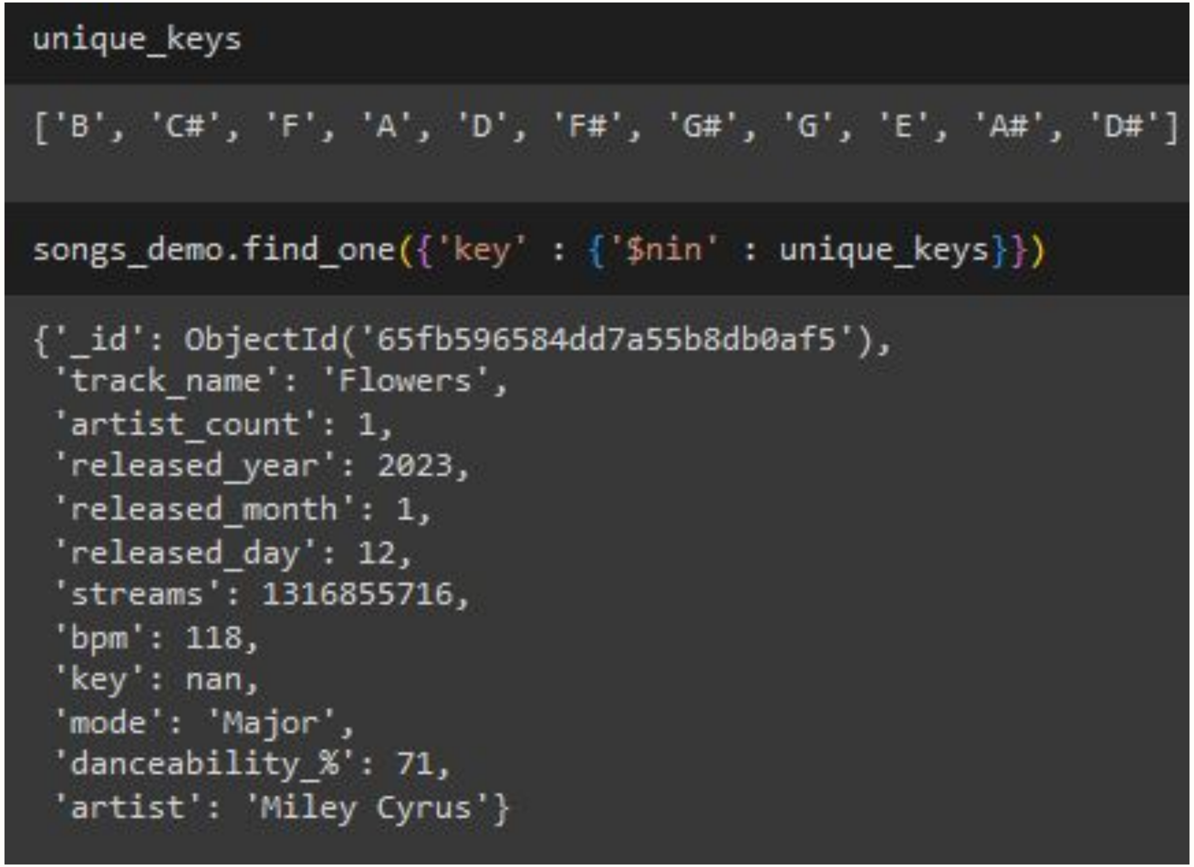
```
380
```

## Null Values

Missing or Null values are a huge issue in real world data work.

How you handle missing data is one of the most important decisions in data management and engineering

For our collection, the ‘key’ field has missing values for some documents


```
unique_keys
```

```
['B', 'C#', 'F', 'A', 'D', 'F#', 'G#', 'G', 'E', 'A#', 'D#']
```

```
songs_demo.find_one({'key' : {'$nin' : unique_keys}})
```

```
{'_id': ObjectId('65fb596584dd7a55b8db0af5'),
 'track_name': 'Flowers',
 'artist_count': 1,
 'released_year': 2023,
 'released_month': 1,
 'released_day': 12,
 'streams': 1316855716,
 'bpm': 118,
 'key': nan,
 'mode': 'Major',
 'danceability_%': 71,
 'artist': 'Miley Cyrus'}
```

# Thanks!

## Do you have any questions?

vravuri@cuhsd.org